Centro Federal de Educação Tecnológica
de Minas Gerais

Algoritmos e Estruturas de Dados

Algoritmos de ordenação

Gabriel Dutra Dias

orientado por
Prof.ª Natália Cosse Batista

Belo Horizonte - MG
2017

Conteúdo

1 Introdução

Algoritmos de ordenação são um problema comum na computação, dado que frequentemente precisamos dispor dados ordenadamente (ordem alfabética de nomes, crescente de idade, etc). Desse modo, existem vários algoritmos de ordenação, onde cada um atende propósitos diferentes.

Neste trabalho apresentarei e compararei alguns dos principais algoritmos de ordenação, considerando algoritmos não-eficientes e eficientes, sendo eles:

1. Não-eficientes
   (a) Bubble Sort
   (b) Selection sort
   (c) Insertion Sort
2. Eficientes
   (a) Quick Sort
   (b) Heap Sort
   (c) Merge sort

2 Implementação

Os algoritmos implementados foram retirados do livro-texto *Projeto de Algoritmos com Implementações em Pascal e C*[1]. A implementação está disponível no GitHub (`https://github.com/gabrieldutra/AlgoritmosOrdenacao`).

O sistema operacional utilizado e recomendado para a implementação e execução foi o Ubuntu 16.04.3, portanto as instruções são válidas apenas para esta distribuição linux e similares.

2.1 Compilação

make - Compila o código
make clean - Apaga todos os arquivos compilados e executáveis

2.2 Execução

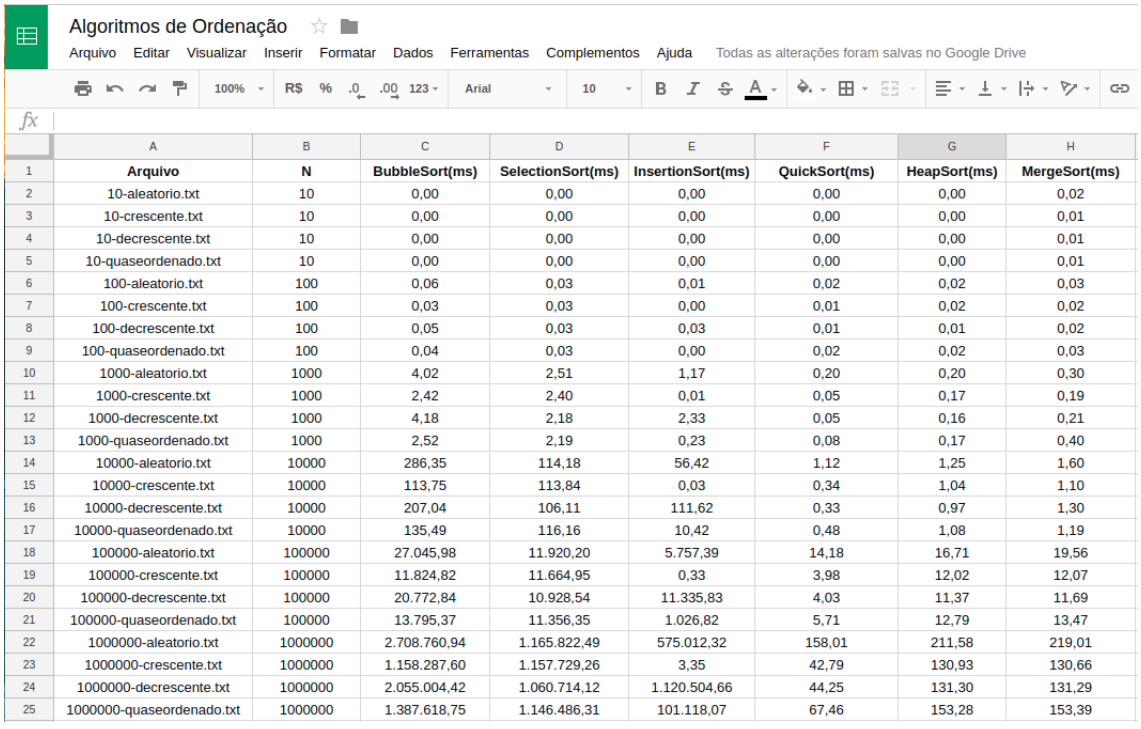
./ordenacao.out [Pasta com os arquivos de teste]

Exemplo: - **./ordenacao.out dataset** executa os testes para todos os arquivos na pasta **dataset**

```
gabriel@NTIC-CII-140:~/AlgoritmosOrdenacao$ ./ordenacao.out dataset
Arquivo N       BubbleSort(ms)  SelectionSort(ms)       InsertionSort(ms)       QuickSort(ms)   HeapSort(ms)    MergeSort(ms)
10-aleatorio.txt        10      0.00    0.00    0.00    0.00    0.00    0.02
10-crescente.txt        10      0.00    0.00    0.00    0.00    0.00    0.01
10-decrescente.txt      10      0.00    0.00    0.00    0.00    0.00    0.01
10-quaseordenado.txt    10      0.00    0.00    0.00    0.00    0.00    0.01
100-aleatorio.txt       100     0.06    0.03    0.01    0.02    0.02    0.03
100-crescente.txt       100     0.03    0.03    0.00    0.01    0.02    0.02
100-decrescente.txt     100     0.05    0.03    0.03    0.01    0.01    0.02
100-quaseordenado.txt   100     0.04    0.03    0.00    0.02    0.02    0.03
1000-aleatorio.txt      1000    4.02    2.51    1.17    0.20    0.20    0.30
1000-crescente.txt      1000    2.42    2.40    0.01    0.05    0.17    0.19
1000-decrescente.txt    1000    4.18    2.18    2.33    0.05    0.16    0.21
1000-quaseordenado.txt  1000    2.52    2.19    0.23    0.08    0.17    0.40
10000-aleatorio.txt     10000   286.35  114.18  56.42   1.12    1.25    1.60
10000-crescente.txt     10000   113.72  113.84  0.03    0.34    1.04    1.10
10000-decrescente.txt   10000   207.04  106.11  111.62  0.33    0.97    1.03
10000-quaseordenado.txt 10000   135.49  116.16  10.42   0.48    1.08    1.19
100000-aleatorio.txt    100000  27045.98        11920.20        5757.39 14.18   16.71   19.56
100000-crescente.txt    100000  11824.82        11664.95        0.33    3.98    12.02   12.07
100000-decrescente.txt  100000  20772.84        10928.54        11335.83        4.03    11.37   11.69
100000-quaseordenado.txt        100000  13795.37        11356.35        1026.82 5.71    12.79   13.47
```

Figura 1: Exemplo de execução

A saída do programa pode ser copiada e colada em alguma planilha. (Nesse exemplo utilizei o Google Planilhas)



	A	B	C	D	E	F	G	H
1	**Arquivo**	**N**	**BubbleSort(ms)**	**SelectionSort(ms)**	**InsertionSort(ms)**	**QuickSort(ms)**	**HeapSort(ms)**	**MergeSort(ms)**
2	10-aleatorio.txt	10	0,00	0,00	0,00	0,00	0,00	0,02
3	10-crescente.txt	10	0,00	0,00	0,00	0,00	0,00	0,01
4	10-decrescente.txt	10	0,00	0,00	0,00	0,00	0,00	0,01
5	10-quaseordenado.txt	10	0,00	0,00	0,00	0,00	0,00	0,01
6	100-aleatorio.txt	100	0,06	0,03	0,01	0,02	0,02	0,03
7	100-crescente.txt	100	0,03	0,03	0,00	0,01	0,02	0,02
8	100-decrescente.txt	100	0,05	0,03	0,03	0,01	0,01	0,02
9	100-quaseordenado.txt	100	0,04	0,03	0,00	0,02	0,02	0,03
10	1000-aleatorio.txt	1000	4,02	2,51	1,17	0,20	0,20	0,30
11	1000-crescente.txt	1000	2,42	2,40	0,01	0,05	0,17	0,19
12	1000-decrescente.txt	1000	4,18	2,18	2,33	0,05	0,16	0,21
13	1000-quaseordenado.txt	1000	2,52	2,19	0,23	0,08	0,17	0,40
14	10000-aleatorio.txt	10000	286,35	114,18	56,42	1,12	1,25	1,60
15	10000-crescente.txt	10000	113,75	113,84	0,03	0,34	1,04	1,10
16	10000-decrescente.txt	10000	207,04	106,11	111,62	0,33	0,97	1,30
17	10000-quaseordenado.txt	10000	135,49	116,16	10,42	0,48	1,08	1,19
18	100000-aleatorio.txt	100000	27.045,98	11.920,20	5.757,39	14,18	16,71	19,56
19	100000-crescente.txt	100000	11.824,82	11.664,95	0,33	3,98	12,02	12,07
20	100000-decrescente.txt	100000	20.772,84	10.928,54	11.335,83	4,03	11,37	11,69
21	100000-quaseordenado.txt	100000	13.795,37	11.356,35	1.026,82	5,71	12,79	13,47
22	1000000-aleatorio.txt	1000000	2.708.760,94	1.165.822,49	575.012,32	158,01	211,58	219,01
23	1000000-crescente.txt	1000000	1.158.287,60	1.157.729,26	3,35	42,79	130,93	130,66
24	1000000-decrescente.txt	1000000	2.055.004,42	1.060.714,12	1.120.504,66	44,25	131,30	131,29
25	1000000-quaseordenado.txt	1000000	1.387.618,75	1.146.486,31	101.118,07	67,46	153,28	153,39

Figura 2: Planilha gerada após a execução

2.3 Gerando arquivos de teste

./gerador.out [Pasta destino] [N] [Tipo de dados]

Tipos de dados: 1 - Ordenados; 2 - Inversamente ordenados; 3 - Quase ordenados; 4 - Aleatórios.

Exemplo: **./gerador.out dataset 100 4** gera um arquivo de teste com 100 números aleatórios na pasta dataset

O nome do arquivo de saída será: [N]-[crescente—decrescente—quaseordenado—aleatorio].txt. No exemplo acima o nome seria: 100-aleatorio.txt.

Obs.: O gerador de teste gera um vetor de tamanho N ordenado (1 até N) e, de acordo com o tipo de dado escolhido, imprime as N linhas do vetor no arquivo destino. No caso de um vetor aleatório ele gera N números aleatórios de 1 a 2N.

2.4 O programa de testes

O programa de teste abre o diretório passado como parâmetro, e para cada arquivo .txt em ordem alfabética executa o seguinte procedimento:

1. Carregar o vetor do arquivo na memória;
2. Para cada um dos algoritmos de ordenação, fazer:
   (a) Copiar o vetor para outra variável;
   (b) Inicia contagem de tempo;
   (c) Executa o algoritmo de ordenação;
   (d) Finaliza a contagem de tempo;
   (e) Confere se o vetor está ordenado (garantindo a funcionalidade do algoritmo);
   (f) Imprime na tela o tempo.

3 Análise de complexidade

3.1 Bubble Sort

O Bubble Sort tem seu pior caso quando o vetor está ordenado inversamente, nessa situação ele tem $f(n)$ definido da seguinte forma: (Algoritmo utilizado disponível no anexo A.1)

$(n - j)$ comparações são feitas $(n - 1)$ vezes, e em todas há troca.

$$f(n) = \sum_{j=1}^{n-1} \sum_{i=1}^{n-j} 1 \tag{1}$$

Resolvendo esses somatórios, chegaremos em:

$$f(n) = \frac{1}{2}(n-1)n \tag{2}$$

Por fim, passando para a notação O:

$$f(n) = O(n^2) \tag{3}$$

3.2 Selection Sort

Semelhante ao Bubble Sort, o algoritmo analisado está na seção A.2.
Complexidade de **comparações**:

$$f(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 \tag{4}$$

O equivalente a:

$$f(n) = \frac{1}{2}(n-1)n \tag{5}$$

Na notação O:

$$f(n) = O(n^2) \tag{6}$$

Para **trocas**, no pior caso sempre ocorre 1 troca a cada iteração do for mais externo, desse modo:

$$f(n) = \sum_{i=1}^{n-1} 1 = n - 1 = O(n) \tag{7}$$

3.3 Insertion Sort

No Insertion Sort o que ocorre é separarmos o vetor em duas partes, onde a primeira está ordenada. A cada iteração "inserimos"o primeiro elemento da segunda parte na primeira parte, de maneira ordenada. Essa inserção é feita comparando o item atual com os itens da primeira parte do vetor, indo do maior para o menor, quando sua posição é encontrada, os outros itens do vetor são deslocados para a direita. Desse modo, o pior caso é quando o vetor está ordenado inversamente, pois a cada iteração serão feitas comparações com todos os elementos da primeira parte e todos serão deslocados. A função de complexidade para comparações e trocas pode ser expressa da seguinte forma:

$$f(n) = \sum_{i=2}^{n} \sum_{j=1}^{i-1} 1 \tag{8}$$

O equivalente a:

$$f(n) = \frac{1}{2}(n-1)n \tag{9}$$

Na notação O:

$$f(n) = O(n^2) \tag{10}$$

3.4 Quick Sort

O pior caso do Quick Sort para comparações é $O(n^2)$. O pior caso ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo já ordenado. Isto faz com que o procedimento Ordena seja chamado recursivamente n vezes, eliminando apenas um item em cada chamada. O Quick Sort, em média é $O(log(n))$[1]

3.5 Heap Sort

O procedimento Refaz gasta cerca de $log(n)$ operações no pior caso, logo o Heap Sort gasta um tempo de execução proporcional a $nlog(n)$, no pior caso.[1]

3.6 Merge Sort

Na contagem de comparações, o comportamento do Mergesort pode ser representado por:

$$T(n) = 2T(n/2) + n - 1, T(1) = 0 \tag{11}$$

$$\vdots$$

$$T(n) = nlog(n) - n + 1 = O(nlog(n)) \tag{12}$$

[1]

3.7 Geral

O programa carrega m arquivos com complexidade n_m. Para cada arquivo, é executado cada um dos algoritmos, como o algoritmo de pior complexidade é $O(n^2)$, esse se sobressai na execução geral. Dessa forma a função que dá o tempo geral de execução é:

$$\sum_{i=1}^{m} O((n_i)^2) \tag{13}$$

4 Testes executados

Foram gerados arquivos de teste com **N** variando entre 1, 10, 100, 1.000, 100.000 e 1.000.000 e para cada valor de N, vetores Aleatórios, Crescentes, Decrescentes e Quase Ordenados.

O computador utilizado nos teste foi um Dell OptiPlex 7010 com um processador Intel Core i5 3a geração com 8 GB de memória RAM. Durante os testes o programa **htop** foi utilizado para monitorar o uso de CPU e memória. Desse modo, garantindo que o computador não iria estourar recursos.

4.1 Tabelas

Tabela 1: Resultados dos algoritmos não-eficientes

Arquivo	N	BubbleSort(ms)	SelectionSort(ms)	InsertionSort(ms)
10-aleatorio.txt	10	0,00	0,00	0,00
10-crescente.txt	10	0,00	0,00	0,00
10-decrescente.txt	10	0,00	0,00	0,00
10-quaseordenado.txt	10	0,00	0,00	0,00
100-aleatorio.txt	100	0,06	0,03	0,01
100-crescente.txt	100	0,03	0,03	0,00
100-decrescente.txt	100	0,05	0,03	0,03
100-quaseordenado.txt	100	0,04	0,03	0,00
1000-aleatorio.txt	1000	4,02	2,51	1,17
1000-crescente.txt	1000	2,42	2,40	0,01
1000-decrescente.txt	1000	4,18	2,18	2,33
1000-quaseordenado.txt	1000	2,52	2,19	0,23
10000-aleatorio.txt	10000	286,35	114,18	56,42
10000-crescente.txt	10000	113,75	113,84	0,03
10000-decrescente.txt	10000	207,04	106,11	111,62
10000-quaseordenado.txt	10000	135,49	116,16	10,42
100000-aleatorio.txt	100000	27.045,98	11.920,20	5.757,39
100000-crescente.txt	100000	11.824,82	11.664,95	0,33
100000-decrescente.txt	100000	20.772,84	10.928,54	11.335,83
100000-quaseordenado.txt	100000	13.795,37	11.356,35	1.026,82
1000000-aleatorio.txt	1000000	2.708.760,94	1.165.822,49	575.012,32
1000000-crescente.txt	1000000	1.158.287,60	1.157.729,26	3,35
1000000-decrescente.txt	1000000	2.055.004,42	1.060.714,12	1.120.504,66
1000000-quaseordenado.txt	1000000	1.387.618,75	1.146.486,31	101.118,07

Tabela 2: Resultados dos algoritmos eficientes

Arquivo	N	QuickSort(ms)	HeapSort(ms)	MergeSort(ms)
10-aleatorio.txt	10	0,00	0,00	0,02
10-crescente.txt	10	0,00	0,00	0,01
10-decrescente.txt	10	0,00	0,00	0,01
10-quaseordenado.txt	10	0,00	0,00	0,01
100-aleatorio.txt	100	0,02	0,02	0,03
100-crescente.txt	100	0,01	0,02	0,02
100-decrescente.txt	100	0,01	0,01	0,02
100-quaseordenado.txt	100	0,02	0,02	0,03
1000-aleatorio.txt	1000	0,20	0,20	0,30
1000-crescente.txt	1000	0,05	0,17	0,19
1000-decrescente.txt	1000	0,05	0,16	0,21
1000-quaseordenado.txt	1000	0,08	0,17	0,40
10000-aleatorio.txt	10000	1,12	1,25	1,60
10000-crescente.txt	10000	0,34	1,04	1,10
10000-decrescente.txt	10000	0,33	0,97	1,30
10000-quaseordenado.txt	10000	0,48	1,08	1,19
100000-aleatorio.txt	100000	14,18	16,71	19,56
100000-crescente.txt	100000	3,98	12,02	12,07
100000-decrescente.txt	100000	4,03	11,37	11,69
100000-quaseordenado.txt	100000	5,71	12,79	13,47
1000000-aleatorio.txt	1000000	158,01	211,58	219,01
1000000-crescente.txt	1000000	42,79	130,93	130,66
1000000-decrescente.txt	1000000	44,25	131,30	131,29
1000000-quaseordenado.txt	1000000	67,46	153,28	153,39

4.2 Gráficos

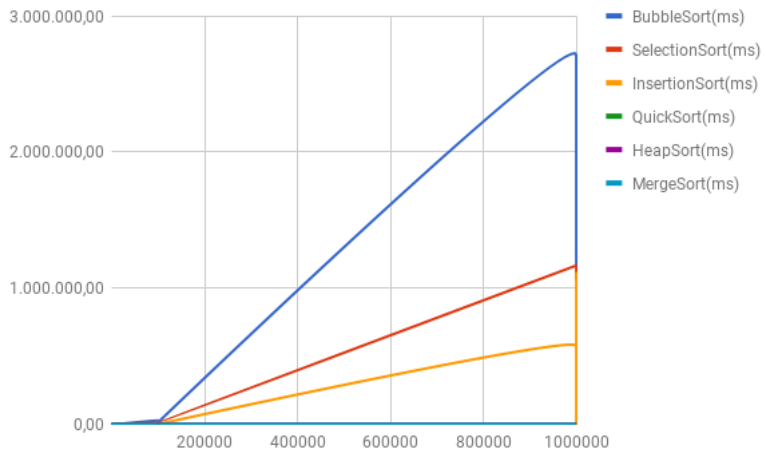


Figura 3: Gráfico Geral

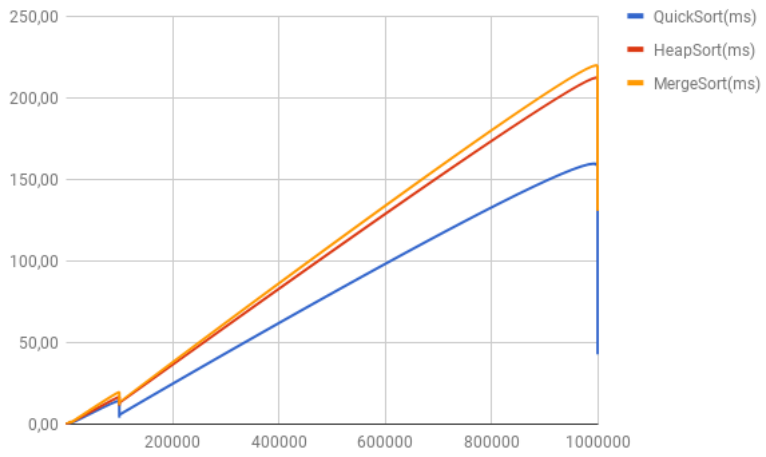


Figura 4: Gráfico Eficientes

4.3 Análises

Entre os algoritmos não-eficientes é fácil perceber que o pior é o Bubble Sort, a diferença entre ele e os outros não-eficientes está no número de trocas que ele faz. Enquanto os outros são $O(n)$ para trocas no pior caso, ele é $O(n^2)$. O melhor foi o Insertion Sort, que faz $O(n)$ comparações e $O(1)$ trocas no melhor caso, tornando-o o melhor dos algoritmos testados quando se trata de um **vetor ordenado**.

Já entre os algoritmos eficientes não há uma disparidade muito grande entre eles, o Merge Sort foi o melhor e o Quick Sort o pior. Todos eles possuem complexidade média $O(nlog(n))$.

5 Conclusão

Considerando todo o trabalho realizado, pode-se concluir que em aplicações que envolvam uma quantidade muito grande de dados (acima de 10.000 por exemplo) já se torna inviável o uso de algoritmos não-eficientes para ordenação.

No entanto, esses algoritmos ainda são muito bons para fins acadêmicos (são em geral fáceis de serem implementados) e atendem a demanda quando se trata de valores pequenos para N.

A Algoritmos não-eficientes

A.1 Bubble Sort

```
/** Bubble Sort
* @param A - Vetor a ser ordenado
* @param n - Tamanho do Vetor
**/
void Bubblesort(TipoVetor A, TipoIndice n)
{
    int i, j;
    int aux;
    for(j=1; j<=n-1; j++){
        for(i=1; i<=n-j; i++){
            if(A[i].Chave > A[i+1].Chave){
                aux = A[i].Chave;
                A[i].Chave = A[i+1].Chave;
                A[i+1].Chave = aux;
            }
        }
    }
}
```

Código 1: Bubble Sort [1]

A.2 Selection Sort

```
/** Selection Sort
* @param A - Vetor a ser ordenado
* @param n - Tamanho do Vetor
**/
void Selecao(TipoVetor A, TipoIndice n)
{ TipoIndice i, j, Min;
  TipoItem x;
  for (i = 1; i <= n - 1; i++)
    { Min = i;
      for (j = i + 1; j <= n; j++)
        if (A[j].Chave < A[Min].Chave) Min = j;
      x = A[Min]; A[Min] = A[i]; A[i] = x;
    }
}
```

Código 2: Selection Sort [1]

A.3 Insertion Sort

```
/** Insertion Sort
* @param A - Vetor a ser ordenado
* @param n - Tamanho do Vetor
**/
void Insercao(TipoVetor A, TipoIndice n)
{ TipoIndice i, j;
  TipoItem x;
  for (i = 2; i <= n; i++)
    { x = A[i];  j = i - 1;
      A[0] = x;  /* sentinela */
      while (x.Chave < A[j].Chave)
        { A[j+1] = A[j];  j--;
        }
      A[j+1] = x;
    }
}
```

Código 3: Insertion Sort [1]

B Algoritmos eficientes

B.1 Quick Sort

```
/** Particao - Rearranja o Vetor A[Esq..Dir] */
void Particao(TipoIndice Esq, TipoIndice Dir, TipoIndice *i, TipoIndice *j,
    TipoVetor A){
     TipoItem x, w;
     *i = Esq;  *j = Dir;
     x = A[(*i + *j) / 2]; /* obtem o pivo x */
     do {
          while (x.Chave > A[*i].Chave) (*i)++;
          while (x.Chave < A[*j].Chave) (*j)--;
          if (*i <= *j){
               w = A[*i]; A[*i] = A[*j]; A[*j] = w;
               (*i)++; (*j)--;
          }
     } while (*i <= *j);
}

/** Ordena - Ordena o Vetor A[Esq..Dir] por meio de particoes */
void Ordena(TipoIndice Esq, TipoIndice Dir, TipoItem *A){
     TipoIndice i, j;
     Particao(Esq, Dir, &i, &j, A);
     if (Esq < j) Ordena(Esq, j, A);
     if (i < Dir) Ordena(i, Dir, A);
}

/** QuickSort - Ordena usando o quicksort
* @param A - Vetor a ser ordenado
* @param n - Tamanho do vetor
*/
void QuickSort(TipoVetor A, TipoIndice n){
     Ordena(1, n, A);
}
```

Código 4: Quick Sort [1]

B.2 Heap Sort

```
/** Refaz - Refaz o Heap nas posicoes passadas */
void Refaz(TipoIndice Esq, TipoIndice Dir, TipoVetor A){
    TipoIndice i = Esq;
    int j;
    TipoItem x;
    j = i * 2;
    x = A[i];
    while (j <= Dir){
        if (j < Dir){
            if (A[j].Chave < A[j+1].Chave) j++;
        }
        if (x.Chave >= A[j].Chave) break;
        A[i] = A[j];
        i = j;  j = i * 2;
    }
    A[i] = x;
}
/** Constroi - Constroi o Heap */
void Constroi(TipoVetor A, TipoIndice n){
    TipoIndice Esq;
    Esq = n / 2 + 1;
    while (Esq > 1){
        Esq--;
        Refaz(Esq, n, A);
    }
}

/** Heapsort - Ordenacao pelo heapsort
* @param A - Vetor a ser ordenado
* @param n - Tamanho do vetor
*/
void Heapsort(TipoVetor A, TipoIndice n){
    TipoIndice Esq, Dir;
    TipoItem x;
    Constroi(A, n);  /* constroi o heap */
    Esq = 1;  Dir = n;
    while (Dir > 1){ /* ordena o vetor */
        x = A[1];  A[1] = A[Dir];  A[Dir] = x;  Dir--;
        Refaz(Esq, Dir, A);
    }
}
```

Código 5: Heap Sort [1]

B.3 Merge Sort

```
/** Merge - intercala A[i..m] e A[m+1..j] em A[i..j]
* @param A - Vetor a ser intercalado
* @param i - Posicao inicial
* @param m - Posicao central
* @param j - Posicao final
*/
void Merge(TipoVetor A, int i, int m, int j){
    TipoVetor B = (TipoVetor) malloc(sizeof(TipoItem)*(MAXTAM+1));
    int x;
    int k = i;
    int l = m+1;

    for (x=i;x<=j;x++) B[x] = A[x];
    x = i;
        while (k<=m && l<=j ) {
        if (B[k].Chave <= B[l].Chave)
            A[x++].Chave = B[k++].Chave;
        else
            A[x++].Chave = B[l++].Chave;
    }

    while (k<=m) A[x++].Chave = B[k++].Chave;

    while (l<=j) A[x++].Chave = B[l++].Chave;

    free(B);

}

/** Mergesort - consiste na intercalacao recursiva de vetores ordenados
* @param A - Vetor a ser ordenado
* @param i - Posicao inicial
* @param j - Posicao final
*/
void Mergesort(TipoVetor A, int i, int j){
    int m;
    if (i < j){
        m = (i + j - 1) / 2;
        Mergesort(A, i, m);
        Mergesort(A, m + 1, j);
        Merge(A, i, m, j); /*Intercala A[i..m] e A[m+1..j] em A[i..j] */
    }
}
```

Código 6: Merge Sort [1]

Referências

[1] N. ZIVIANI. *Projeto de Algoritmos com Implementações em Pascal e C. 2 ed.* Cengage Learning, 2004.